Lecture 7 – Mutation Testing

AAA705: Software Testing and Quality Assurance

Jihyeok Park

2024 Spring

Recall

- **Symbolic Execution**
  - Basic Idea
  - Satisfiability Modulo Theories (SMT)
  - Limitations of Symbolic Execution
- **Dynamic Symbolic Execution (DSE)**
  - Search Heuristics
  - Example – Hash Function
  - Example – Loops
  - Example – Data Structures
  - Realistic Implementation
- **Other Hybrid Analysis Techniques**

Contents

Contents

Mutation Testing

PLRG

- **Mutation testing** is a white-box and fault-based testing technique.
- **Inverts** the testing adequacy criterion: the goal is to **access** the effectiveness of the existing test suite in terms of its **fault detection capabilities**.
  - **Test suites** test **programs**
  - **Mutants** test **test suites**
- The most widely used adequacy score is **mutation score**: it measures the quality of the given test suite as **the percentage of injected faults that you can detect**.

Mutation Testing

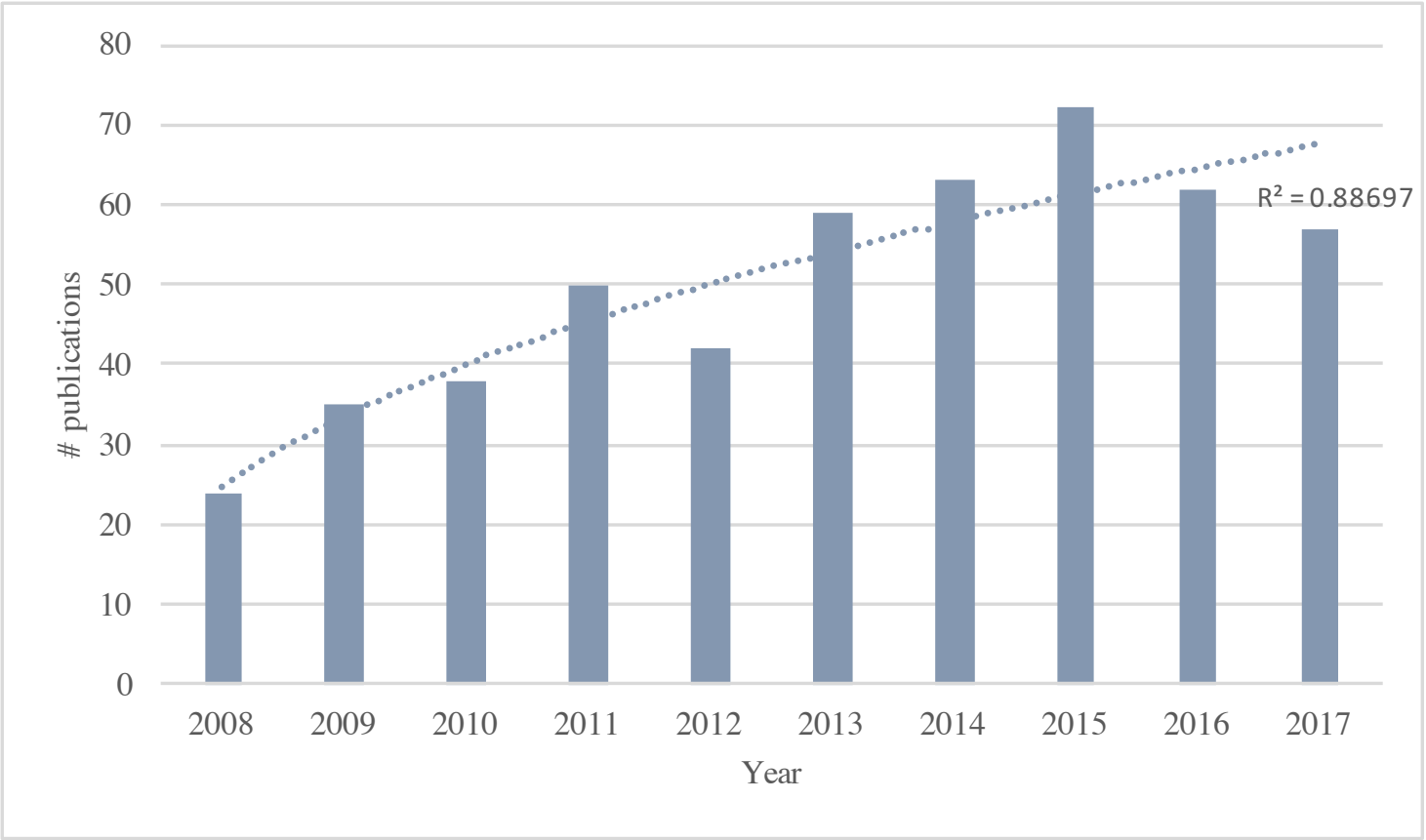


Figure 1: Number of mutation testing publications per year (years: 2008-2017).

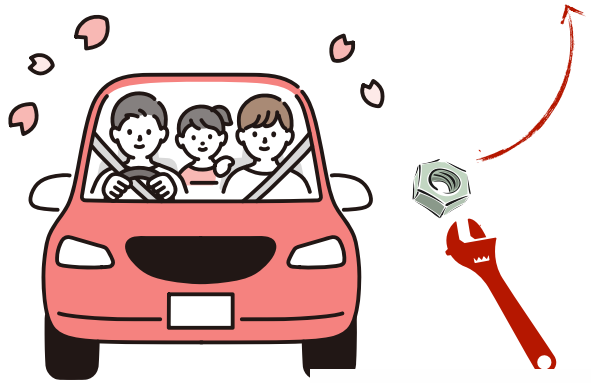
M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, M. Harman, Mutation Testing Advances: An Analysis and Survey, Advances in Computers, 2017.

Mutation Testing

Which environment is better test environment for car? **Why?**

Mutation Testing

Let's **sabotage** the car! In the **bendy road**, the car will **crash**!



Mutation Testing

PLRG

- Testing is a sampling process: without a priori knowledge of faults, it is difficult to access how well a technique samples.

- **Mutation testing**: the **quality** of a test suite can be indirectly measured by **artificially injecting faults** and **checking how well the test suite can detect them**.

  - Seed the original implementations with faults (the seeded version are called **mutants**

  - **Execute** the given test suite

  - If we get different test results, the introduced faults (the mutant) has been identified (i.e., the mutant is **killed**). If not, the mutant is still **alive**.

Fundamental Hypotheses

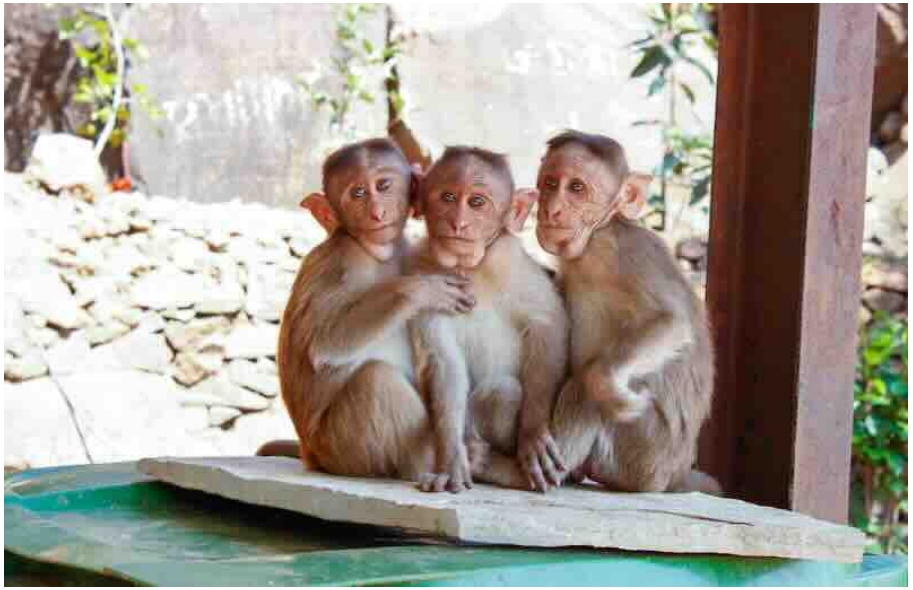
PLRG

- **Mutation testing** is based on two **fundamental hypotheses**:

1. **Competent Programmer Hypothesis**

2. **Coupling Effect Hypothesis**

(1) Competent Programmer Hypothesis



What do the **programmers** and the **monkeys** have in common when they write programs?

They both write **buggy code**!

(1) Competent Programmer Hypothesis

On average, programmers are **competent** (they write **almost correct** programs). A **faulty program** source code is different from the correct one **only in a few**, minor detail.

(2) Coupling Effect Hypothesis

PLRG

- If a **test suite** detects **all small syntactic faults**, it will also detect **larger, semantics faults**: especially if those semantic faults are **coupled with** the small faults.

  - Richard A. DeMillo and Richard J. Lipton and Frederick Gerald Sayward, Hints on Test Data Selection: Help for the Practicing Programmer, Computer, 11(4), 1978.

  - A. Jefferson Offutt, Investigations of the Software Testing Coupling Effect , ACM Transactions on Software Engineering and Methodology, 1(1), January 1992.

Fundamental Hypotheses

PLRG

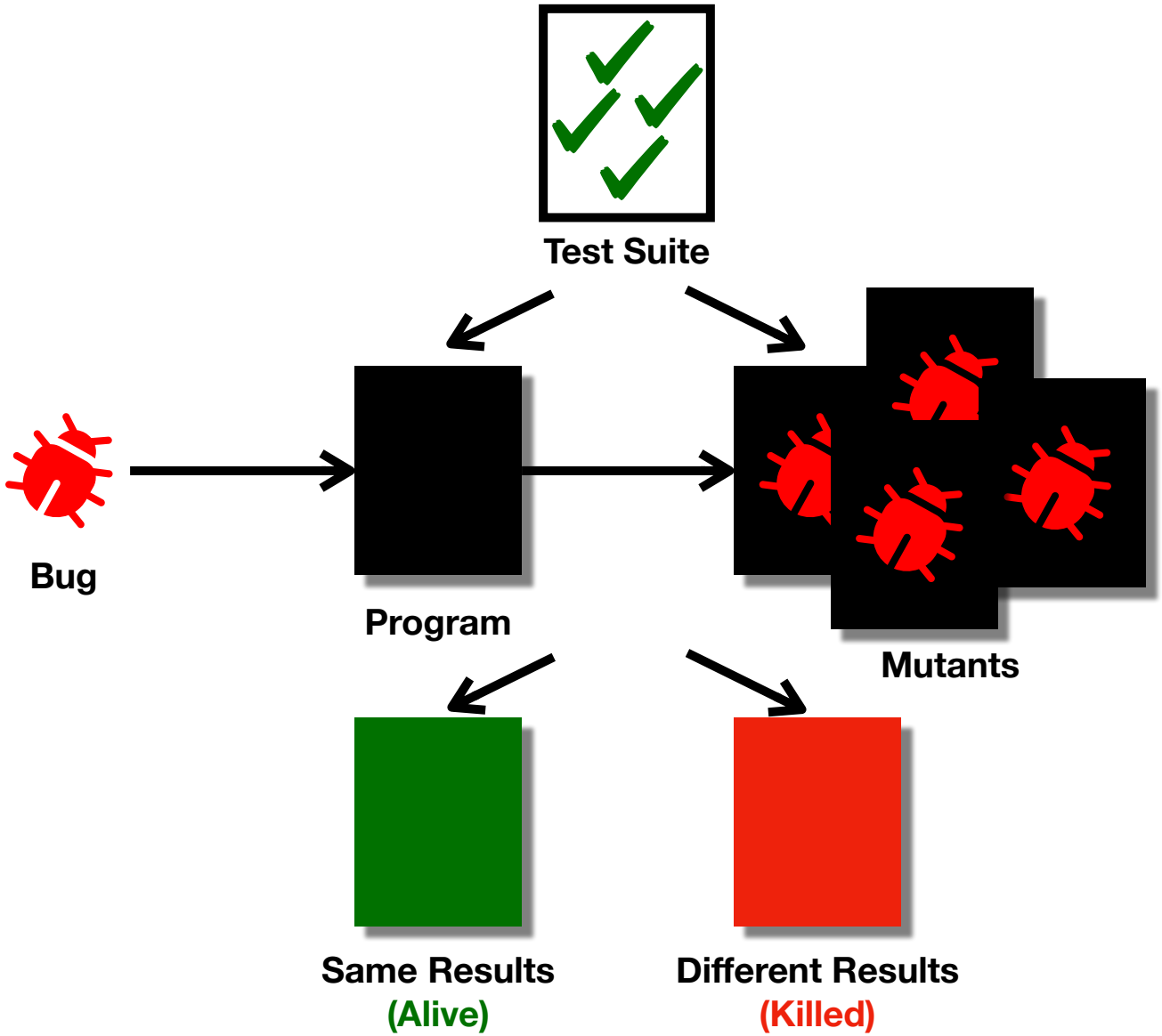
- **Mutation testing** is based on two **fundamental hypotheses**:

  1. **Competent Programmer Hypothesis**: programmers are likely to make simple faults.

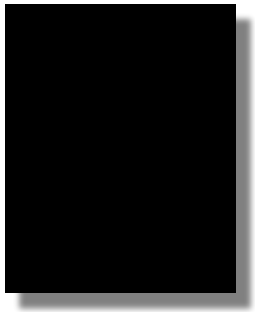
  2. **Coupling Effect Hypothesis**: if we catch all the simple faults, we will be able to catch more complicated faults.

- Let's artificially **inject simple faults**!

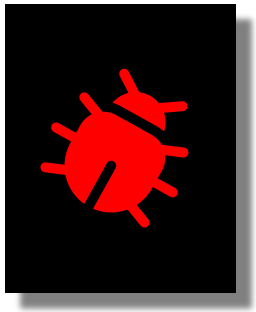
Overall Process



Mutation Generation



Program P



Mutant P'

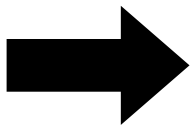
- P' differs from P by a **simple mutation**.
- **Mutation**: a typical simple error programmers are likely to make – off-by-one errors, typo, mistaken identity, etc.

Mutation Operator

An atomic rule that is used to generate a mutant

ABS: Absolute Value Insertion

```
x = 4 * y;
```



```
x = 4 * abs(y);
x = 4 * -abs(y);
x = 4 * failOnZero(y);
```

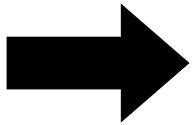
A Fortran Language System for Mutation-Based Software Testing, Kim N. King and Jeff Offutt. Software Practice and Experience, 21(7):686-718, July 1991

Mutation Operator

An atomic rule that is used to generate a mutant

AOR: Arithmetic Operator Replacement

```
x = y + z;
```



```
x = y * z;
x = y - z;
x = y / z;
```

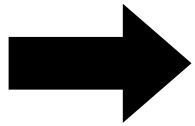
A Fortran Language System for Mutation-Based Software Testing, Kim N. King and Jeff Offutt. Software Practice and Experience, 21(7):686-718, July 1991

Mutation Operator

An atomic rule that is used to generate a mutant

ROR: Relational Operator Replacement

```
if (x >= y)
```



```
if (x > y)
if (x == y)
if (x < y)
if (x != y)
```

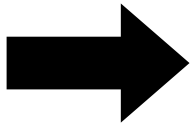
A Fortran Language System for Mutation-Based Software Testing, Kim N. King and Jeff Offutt. Software Practice and Experience, 21(7):686-718, July 1991

Mutation Operator

An atomic rule that is used to generate a mutant

COR: Conditional Operator Replacement

```
if (x && y)
```



```
if (x || y)
if (x & y)
if (x | y)
```

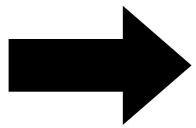
A Fortran Language System for Mutation-Based Software Testing, Kim N. King and Jeff Offutt. Software Practice and Experience, 21(7):686-718, July 1991

Mutation Operator

An atomic rule that is used to generate a mutant

SDL: Statement Deletion

```
x = 3;
y = x + 5;
z = x - y;
```



```
x = 3;

z = x - y;
```

A Fortran Language System for Mutation-Based Software Testing, Kim N. King and Jeff Offutt. Software Practice and Experience, 21(7):686-718, July 1991

Mutation Operator

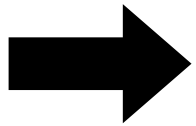
- **Any systematic and syntactic change operator** can be considered.
- For **C**: 71 Mutation Operators – **Statement** 15, **Operator** 46, **Variable** 7, **Constant** 3
  - Design of Mutant Operators for the C Programming Language by Hiralal Agrawal, Richard A DeMillo, R Hathaway,William Hsu,Wynne Hsu, Edward W Krauser, Rhonda J Martin, Aditya P Mathur, Eugene H Spafford, technical report, Purdue University, 1989
- For **Java**: 24 Mutation Operators – **Access Control** 1, **Inheritance** 7, **Polymorphism** 4, **Overloading** 4, **Java-Specific Features** 4, **Common Programming Mistakes** 4
  - Y.-S. Ma, Y.-R. Kwon, and J. Offutt. Inter-class mutation operators for java. In Proceedings of the 13th International Symposium on Software Reliability Engineering, ISSRE '02, pages 352–, Washington, DC, USA, 2002. IEEE Computer Society.
- For **Spreadsheets**
  - R. Abraham and M. Erwig. Mutation operators for spreadsheets. IEEE Transactions on Software Engineering, 35(1):94–108, 2009.

Mutation Operator – Java

For **Java** – 1 mutation for **Information Hiding (Access Control)**

AMC: Access Modifier Change

```
public Stack s;
```



```
private Stack s;
protected Stack s;
Stack s;
```

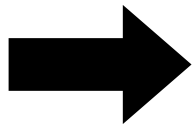
Y.-S. Ma, Y.-R. Kwon, and J. Offutt. Inter-class mutation operators for java. In Proceedings of the 13th International Symposium on Software Reliability Engineering, ISSRE '02, pages 352–, Washington, DC, USA, 2002. IEEE Computer Society.

Mutation Operator – Java

For **Java** – 7 mutations for **Inheritance**

IHD: Hiding Variable Deletion

```
class List {
  int size;
  ...
}
class Stack extends List {
  int size;
  ...
}
```



```
class List {
  int size;
  ...
}
class Stack extends List {

  ...
}
```

Y.-S. Ma, Y.-R. Kwon, and J. Offutt. Inter-class mutation operators for java. In Proceedings of the 13th International Symposium on Software Reliability Engineering, ISSRE '02, pages 352–, Washington, DC, USA, 2002. IEEE Computer Society.

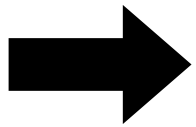
Mutation Operator – Java

For **Java** – 7 mutations for **Inheritance**

IHI: Hiding Variable Insertion

```
class List {
  int size;
  ...
}
class Stack extends List {

  ...
}
```



```
class List {
  int size;
  ...
}
class Stack extends List {
  int size;
  ...
}
```

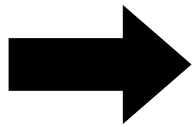
Y.-S. Ma, Y.-R. Kwon, and J. Offutt. Inter-class mutation operators for java. In Proceedings of the 13th International Symposium on Software Reliability Engineering, ISSRE '02, pages 352–, Washington, DC, USA, 2002. IEEE Computer Society.

Mutation Operator – Java

For **Java** – 7 mutations for **Inheritance**

IOD: Overriding Method Deletion

```
class List {
  Push (int a) { ... }
  ...
}
class Stack extends List {
  Push (int a) { ... }
  ...
}
```



```
class List {
  Push (int a) { ... }
  ...
}
class Stack extends List {

  ...
}
```

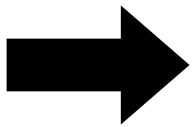
Y.-S. Ma, Y.-R. Kwon, and J. Offutt. Inter-class mutation operators for java. In Proceedings of the 13th International Symposium on Software Reliability Engineering, ISSRE '02, pages 352–, Washington, DC, USA, 2002. IEEE Computer Society.

Mutation Operator – Java

For **Java** – 7 mutations for **Inheritance**

IOP: Overriden Method Calling Position Change

```
class List {
  void SetEnv() {
    size = 5; ... }
  ...
}
class Stack extends List {
  void SetEnv() {
    super.SetEnv();
    size = 10;
  }
  ...
}
```



```
class List {
  void SetEnv() {
    size = 5; ... }
  ...
}
class Stack extends List {
  void SetEnv() {
    size = 10;
    super.SetEnv();
  }
  ...
}
```

Y.-S. Ma, Y.-R. Kwon, and J. Offutt. Inter-class mutation operators for java. In Proceedings of the 13th International Symposium on Software Reliability Engineering, ISSRE '02, pages 352–, Washington, DC, USA, 2002. IEEE Computer Society.

Mutation Operator – Java

For **Java** – 4 mutations for **Polymorphism**

For **Java** – 4 mutations for **Overloading**

For **Java** – 4 mutations for **Java-Specific Features**

- (e.g., `this`, `static`, member variable initialization, default constructor)

For **Java** – 4 mutations for **Common Programming Mistakes**

Y.-S. Ma, Y.-R. Kwon, and J. Offutt. Inter-class mutation operators for java. In Proceedings of the 13th International Symposium on Software Reliability Engineering, ISSRE '02, pages 352–, Washington, DC, USA, 2002. IEEE Computer Society.

Mutation Operator – Spreadsheets

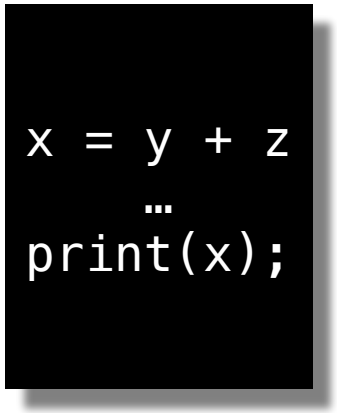
PLRG

- **Any systematic and syntactic change operator** can be considered.
- For **spreadsheets**[1]:
  - RCR (Reference for Constant Replacement)
  - FRC (Formula Replacement with Constants)
  - CRE (Contiguous Range Expansion)
  - CRS (Contiguous Range Shrinking)
  - etc.

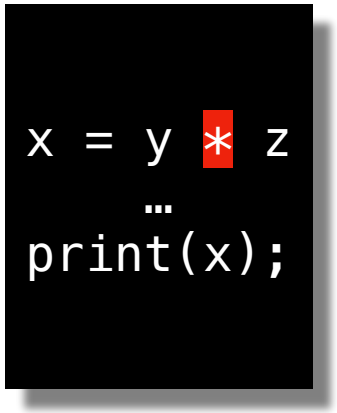
[1] R. Abraham and M. Erwig. Mutation operators for spreadsheets. IEEE Transactions on Software Engineering, 35(1):94–108, 2009.

Kill vs Alive

PLRG



Program P



Mutant P'

Test: (y, z) = (2, 2)	4	4	**Alive**
Test: (y, z) = (3, 1)	4	3	**Kill**

Kill vs Alive

```
x = y + y
    …
print(x);
```

Program P

```
x = y * 2
    …
print(x);
```

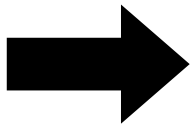
Mutant P'

Test: y = 2	4	4	**Alive**
Test: y = 3	6	6	**Alive**

Equivalent Mutants

- What if a **mutant** has the same **behavior** as the original program?
- For example, consider the following program and its mutant:

```
x = y + y;
```



```
x = y * 2;
```

- Checking whether an arbitrary mutant is equivalent or not is **undecidable**.
- This is one of the **major obstacles** to the mainstream adoption of mutation testing.
  - My **mutation score** is 70%. Is my test suite bad, or are there too many equivalent mutants?

Equivalent Mutants

$$MS = \frac{(\#\text{ of killed mutants})}{(\#\text{ of non-equivalent mutants})}$$

$$MS = \frac{(\#\text{ of killed mutants})}{(\#\text{ of all mutants})}$$

How to Kill A Mutant

Three conditions need to be satisfied to kill a mutant:

- **Reachability**: your test execution needs to reach (i.e., cover) the mutant.
- **Infection**: the mutated code should infect the program state (i.e., the value of the mutated expression differs from the value of the original expression).
- **Propagation**: the infected state should propagate to the observable state.

We categorize the kill conditions into two types: **weak** and **strong**.

- (**Weak Kill**) = **Reachability** + **Infection**
  (i.e., we stop after confirming infection, do not check the propagation to the outside world)
- (**Strong Kill**) = **Reachability** + **Infection** + **Propagation**
  (i.e., the kill can be observed from the outside world)

How to Kill A Mutant

```
if (x < y) {
  if (z < y) {
//if (z < y + 1) {
    if (x < z) {
      result = z;
    } else {
      result = x;
    }
  } else {
    result = y;
  }
} else {
  result = 0;
}
```

- Reachability Condition:

  `x < y`

- Infection Condition:

  `(z < y) != (z < y + 1)`

- **Weak Kill Condition**:

  `(x < y) && ((z < y) != (z < y + 1))`

  or simply `(x < y) && (z == y)`

- Propagation Condition:

  $y = z$

- **Strong Kill Condition**:

  `(x < y) && (z == y) && (y != z)`

Scalability

- **Normal testing**: 1 program $\times$ 100 test cases
- **Mutation testing**: 1 program $\times$ 10000 mutants (**including compilation!**) $\times$ 100 test cases
- We tend to get a large number of mutants:
  - No prior knowledge of **which mutation** operator is the **most effective** (w.r.t. improving the test suite quality): the default is to apply everything
  - **Programs are large**!

Scalability – Do Fewer

- **Mutation Sampling** – generate a large number of mutants, but use only a **subset** of them (natural question: how to do we select?)
- **Subsuming Mutant** – a mutant P' **subsumes** another mutant P'' if and only if killing P' implies killing P''.
  - True subsumption relationship is **undecidable**.
  - **Dynamic subsumption** is defined w.r.t. a given test suite.
  - **Static subsumption** is defined with results of static analysis.
- **Selective Mutation** – apply only a **subset** of mutation **operators**.

Scalability – Do Smarter

- **Super-mutant** – **compile** all mutants into a **single program**, then, activate a specific subset at the runtime (saves the compilation time).
- **Weak mutation testing** – **relax the kill criterion to weak kills** (requires instrumentation for the embedded oracle).
- **Parallel/distributed mutation testing** – obvious.

Scalability – Trivial Compiler Equivalence (TCE)

PLRG

- **Trivial Compiler Equivalence (TCE)**[2]– a mutant is **trivially equivalent** if the binary code of the mutant is **same** as the binary code of the original program after the compilation (thanks to the **compiler optimization**).

- A large scale empirical study showed that TCE can detect **7% of the mutants to be equivalent**; more importantly, **21% of all mutants were duplicates** (i.e., not equivalent to the original program, but identical to another non-equivalent mutant).

[2] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In Proceedings of the 37th International Conference on Software Engineering-Volume 1, pages 936–946. IEEE Press, 2015.

Higher Order Mutants

PLRG

- **First Order Mutants (FOM)** – a single mutation operator is applied to the original program.
- **Higher Order Mutants (HOM)** – multiple mutation operators are applied to the original program.
- Some studies claim that, while most of the FOMs are trivial to kill, few of them are coupled with real faults.
- We can **search** for a combination of FOMs that results in a **hard-to-kill** HOM.

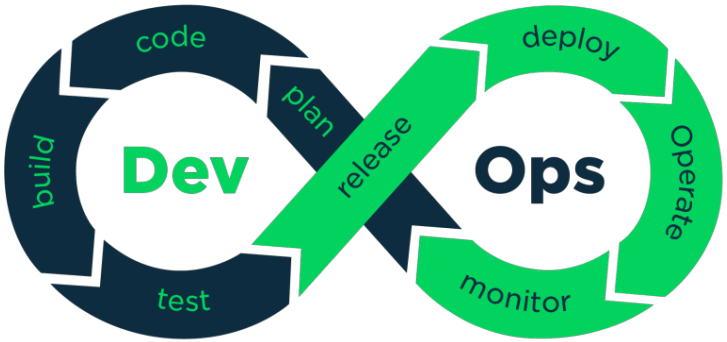
Tools

Diverse **mutation testing tools** are available:

- **Fortran** – **Mothra** (a long-lasting impact on the mutation testing)
- **C/C++** – Proteum, MiLU, MUSIC
- **Java** – muJava, Major, Javalanche, PIT
- **JavaScript** – Stryker
- **Ruby** – Heckle
- **Python** – Mutatest

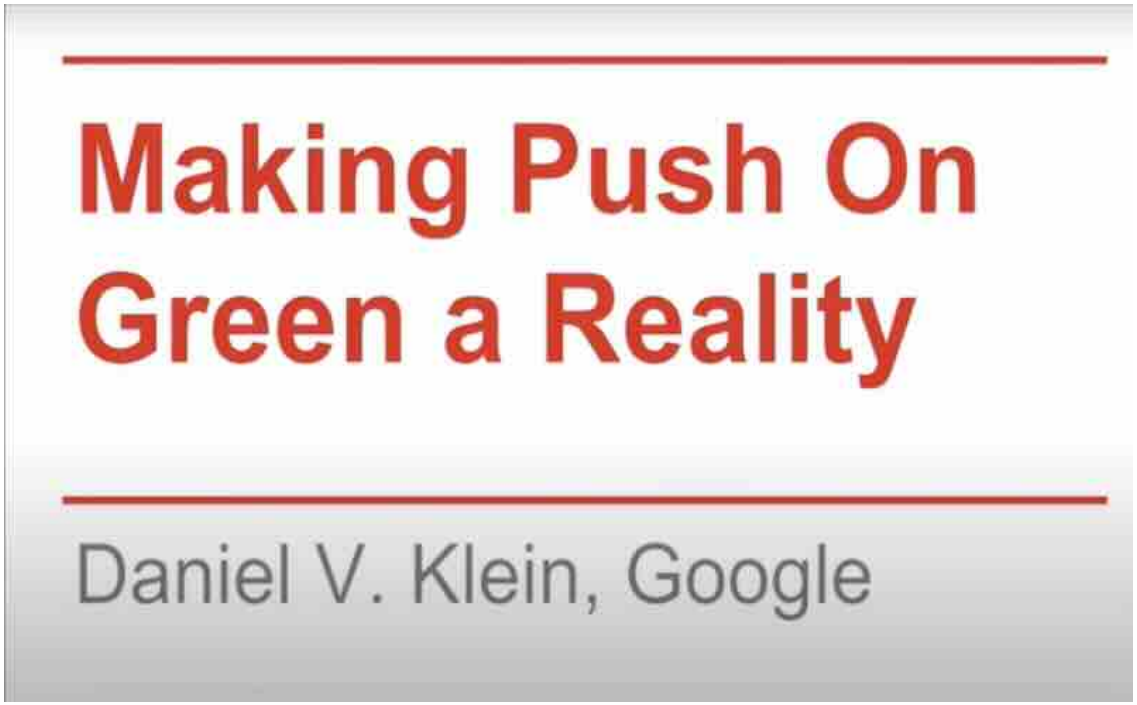
Contents

Push On **Green**



- A **DevOps** concept popularized by Google, more commonly and also known as: **Continuous Integration and Deployment (CI/CD)**
- Newest version of software is automatically deployed whenever all tests pass.
- Test results are critical
  - **False Negative** (i.e., test passes even though there is a bug): you end up releasing a buggy software
  - **False Positive** (i.e., test fails even though there is no bug): slows down the development process

Push On **Green**



Making "Push On Green" a Reality: Issues and Actions Involved in Maintaining a Production Service[3]

(LISA stands for Large Installation System Administration Conference)

[3] A USENIX LISA 2014 presentation given by Daniel Klein, Google: https://www.usenix.org/conference/lisa14/conference-program/presentation/klein

Test Flakiness

- We call a test case to be **flaky** if it changes outcome against the same codebase.

- This creates a **huge problem** for Pass on Green philosophy: when a test transitions from pass to fail, is it flaky or is it actually a real problem?

Test Flakiness

Analysis of Test Results at Google

- Analysis of a large sample of tests (1 month) showed:
  - 84% of transitions from Pass -> Fail are from "flaky" tests
  - Only 1.23% of tests ever found a breakage
  - Frequently changed files more likely to cause a breakage
  - 3 or more developers changing a file is more likely to cause a breakage
  - Changes "closer" in the dependency graph more likely to cause a breakage
  - Certain people / automation more likely to cause breakages (oops!)
  - Certain languages more likely to cause breakages (sorry)

Google

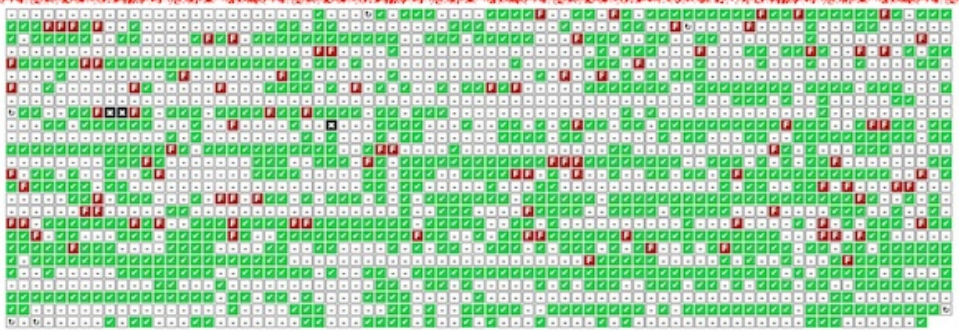
See: prior deck about Google CI System, See this paper about piper and CLs

"The State of Continuous Integration Testing at Google", John Micco, ICST 2017 Keynote (https://research.google/pubs/pub45880/)

Test Flakiness

Flaky Tests

- Test Flakiness is a huge problem
- Flakiness is a test that is observed to both Pass and Fail with the same code
- Almost 16% of our 4.2M tests have some level of flakiness
- Flaky failures frequently block and delay releases
- Developers ignore flaky tests when submitting - sometimes incorrectly
- We spend between 2 and 16% of our compute resources re-running flaky tests



Google

"The State of Continuous Integration Testing at Google", John Micco, ICST 2017 Keynote (https://research.google/pubs/pub45880/)

Sources of Flakiness

PLRG

- **Parallelism** – tests are run in parallel
- **Timeouts** – tests that take too long to run
- **State Management** – tests that depend on the state of the system
- **Data Management** – tests that depend on the data
- **Algorithm** – Non-deterministic algorithms

Solutions?

- Better Synchonization
- Thread-safe code + independent execution environment
- Break-down long sequences + step-wise synchronization
- Explicit pre-condition setup for both state and data + avoid dependencies between test executions
- Fixed seed for random number generation

Your Tests Aren't Flaky

PLRG

- A talk given by Alister Scott (Automattic) at GTAC 2015 (GTAC – Google Test Automation Conference)
  - https://www.youtube.com/watch?v=hmk1h40shaE
  - Here is the **slide**
- "That test is falky" **is not** a get out of jail free card
- A re-run culture is **toxic**

Research on Test Flakiness

- **Detection** – is this test failure real, or a result of flakiness?
- **Prediction** – how likely is this test case to be flaky?
- **Repair** – automatically remove flakiness? (Most ambitious goal)

Detection of Flaky Tests

- A test fails. How do you determine whether it is flaky or not?
- A test case that transitions from pass to fail but does not cover any of the changed part is likely to be flaky! (because the changed behavior is caused by the changed code)
- DeFlaker: Automatically Detecting Flaky Tests, Jonathan Bell; Owolabi Legunsen; Michael Hilton; Lamyaa Eloussi; Tifany Yung; Darko Marinov, ICSE 2018 (https://ieeexplore.ieee.org/abstract/document/8453104)

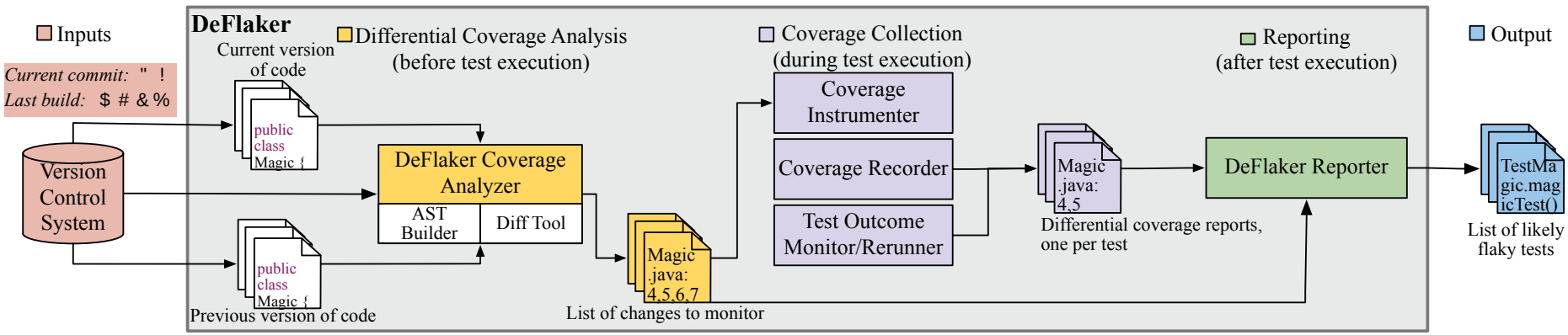


Figure 1: High-level architecture of DeFlaker, with three phases: before, during and after test execution.

Detection of Flaky Tests

Detected 4,846 flaky tests in 26 open-source projects

Table 1: Number of flaky tests found by re-running 5,966 builds of 26 open-source projects. We consider only new test failures where a test passed on the previous commit, and report flakes reported by each phase of our RERUN strategies. DEFLAKER found more flaky tests than the Surefire or Fork rerun strategies: only the very costly Reboot strategy found more flaky tests than DEFLAKER.

		Test Methods in Project		Total New	Confirmed flaky by RERUN strategy			DEFLAKER labeled as: Flaky		DEFLAKER labeled as: Not Flaky	
Project	**#SHAs**	**Total**	**Failing**	**Failures**	**Surefire**	**+Fork**	**++Reboot**	**Confirmed**	**Unconf.**	**Confirmed**	**Unconf.**
achilles	227	337	77	242	13	14	230	225	4	5	8
ambari	500	896	7	75	52	71	74	74	0	0	1
assertj-core	29	6,261	2	3	2	2	2	2	0	0	1
checkstyle	500	1,787	1	1	0	0	0	0	0	0	1
cloudera.oryx	332	275	23	29	5	5	5	5	20	0	4
commons-exec	70	89	2	22	22	22	22	21	0	1	0
dropwizard	298	428	1	60	60	60	60	55	0	5	0
hadoop	298	2,361	365	1,081	284	865	1,054	1,028	25	26	2
handlebars	27	712	7	9	3	7	7	6	2	1	0
hbase	127	431	106	406	62	242	390	383	12	7	4
hector	159	142	12	87	0	74	79	72	4	7	4
httpcore	34	712	2	2	2	2	2	1	0	1	0
jackrabbit-oak	500	4,035	26	34	10	33	34	32	0	2	0
jimfs	164	628	7	21	21	21	21	15	0	6	0
logback	50	964	11	18	18	18	18	18	0	0	0
ninja	317	307	37	122	37	77	110	94	2	16	10
okhttp	500	1,778	129	333	296	305	310	231	0	79	23
oozie	113	1,025	1,065	2,246	42	2,032	2,244	2,234	0	10	2
orbit	227	86	9	86	84	85	85	73	0	12	1
oryx	212	200	38	46	14	14	46	14	0	32	0
spring-boot	111	2,002	67	140	73	107	135	135	3	0	2
tachyon	500	470	4	5	3	5	5	5	0	0	0
togglz	140	227	21	28	5	14	28	28	0	0	0
undertow	7	340	0	0	0	0	0	0	0	0	0
wro4j	306	1,160	114	217	39	96	99	80	8	19	110
zxing	218	415	2	15	15	15	15	15	0	0	0
26 Total	5,966	28,068	2,135	5,328	1,162	4,186	5,075	4,846	80	229	173

Prediction of Flaky Tests

- Can we build a predictive model that can tell us whether a test case is likely to be flaky?
- One possible approach is to **collect features** of known flaky tests and perform **supervised learning** to predict flakiness.
- "FlakeFlagger: Predicting Flakiness Without Rerunning Tests", Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell, ICSE 2021 (https://ieeexplore.ieee.org/abstract/document/9402098)

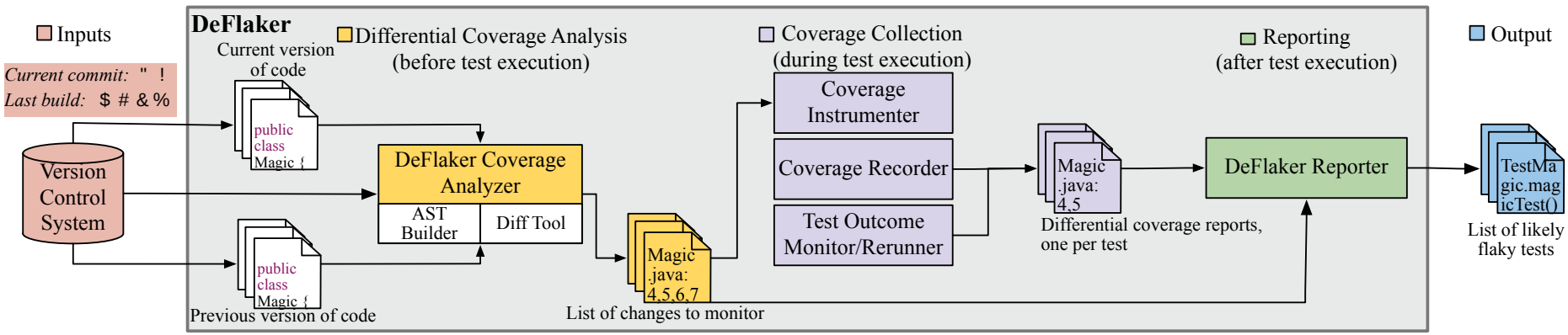


Figure 1: High-level architecture of DeFlaker, with three phases: before, during and after test execution.

Prediction of Flaky Tests

PLRG

It utilizes the following features for supervised learning:

	Feature	Description
Test Smells	Indirect Testing	True if the test interacts with the object under test via an intermediary [24]
	Eager Testing	True if the test exercises more than one method of the tested object [24]
	Test Run War	True if the test allocates a file or resource which might be used by other tests [24]
	Conditional Logic	True if the test has a conditional if-statement within the test method body [25]
	Fire and Forget	True if the test launches background threads or tasks. [26]
	Mystery Guest	True if the test accesses external resources [24]
	Assertion Roulette	True if the test has multiple assertions [24]
	Resources Optimism	True if the test accesses external resources without checking their availability [24]
Numeric Features	Test Lines of Code	Number of lines of code in the test method body
	Number of Assertions	Number of assertions checked by the test
	Execution Time	Running time for the test execution
	Source Covered Lines	Number of lines covered by each test, counting only production code
	Covered Lines	Total number of lines of code covered by the test
	Source Covered Classes	Total number of production classes covered by each test
	External Libraries	Number of external libraries used by the test
	Covered Lines Churn	h-index capturing churn of covered lines in past 5, 10, 25, 50, 75, 100, 500, and 10,000 commits. Each value h indicates that at least h lines were modified at least h times in that period.

Prediction of Flaky Tests

PLRG

It has 86% prediction accuracy for flaky tests on 23 projects

		Flaky by	FlakeFlagger							Vocabulary-Based Approach [12]							Combined Approach						
Project	**Tests**	**Reruns**	**TP**	**FN**	**FP**	**TN**	**Pr**	**R**	**F**	**TP**	**FN**	**FP**	**TN**	**Pr**	**R**	**F**	**TP**	**FN**	**FP**	**TN**	**Pr**	**R**	**F**
spring-boot	2,108	160	139	21	15	1,933	90%	87%	89%	134	26	703	1,245	16%	84%	27%	143	17	18	1,930	89%	89%	89%
hbase	431	145	129	16	32	254	80%	89%	84%	89	56	152	134	37%	61%	46%	130	15	33	253	80%	90%	84%
alluxio	187	116	116	0	0	71	100%	100%	100%	108	8	11	60	91%	93%	92%	116	0	0	71	100%	100%	100%
okhttp	810	100	52	48	159	551	25%	52%	33%	79	21	444	266	15%	79%	25%	46	54	104	606	31%	46%	37%
ambari	324	52	47	5	3	269	94%	90%	92%	36	16	121	151	23%	69%	34%	47	5	3	269	94%	90%	92%
hector	142	33	30	3	8	101	79%	91%	85%	13	20	23	86	36%	39%	38%	25	8	11	98	69%	76%	72%
activiti	2,043	32	10	22	43	1,968	19%	31%	24%	12	20	531	1,480	2%	38%	4%	7	25	34	1,977	17%	22%	19%
java-websocket	145	23	19	4	1	121	95%	83%	88%	23	0	74	48	24%	100%	38%	19	4	4	118	83%	83%	83%
wildfly	1,023	23	11	12	27	973	29%	48%	36%	20	3	554	446	3%	87%	7%	17	6	24	976	41%	74%	53%
httpcore	712	22	14	8	23	667	38%	64%	47%	16	6	375	315	4%	73%	8%	15	7	24	666	38%	68%	49%
logback	805	22	3	19	17	766	15%	14%	14%	10	12	259	524	4%	45%	7%	5	17	11	772	31%	23%	26%
incubator-dubbo	2,174	19	8	11	35	2,120	19%	42%	26%	11	8	813	1,342	1%	58%	3%	13	6	23	2,132	36%	68%	47%
http-request	163	18	12	6	6	139	67%	67%	67%	16	2	84	61	16%	89%	27%	12	6	6	139	67%	67%	67%
wro4j	1,135	16	4	12	2	1,117	67%	25%	36%	2	14	101	1,018	2%	12%	3%	0	16	1	1,118	0%	0%	0%
orbit	86	7	1	6	8	71	11%	14%	12%	6	1	32	47	16%	86%	27%	1	6	7	72	12%	14%	13%
undertow	183	7	2	5	8	168	20%	29%	24%	6	1	63	113	9%	86%	16%	3	4	8	168	27%	43%	33%
achilles	1,317	4	2	2	3	1,310	40%	50%	44%	0	4	0	1,313	0%	0%	0%	0	4	0	1,313	0%	0%	0%
elastic-job-lite	558	3	0	3	0	555	0%	0%	0%	0	3	34	521	0%	0%	0%	1	2	0	555	100%	33%	50%
zxing	345	2	0	2	2	341	0%	0%	0%	1	1	144	199	1%	50%	1%	0	2	2	341	0%	0%	0%
assertj-core	6,261	1	0	1	5	6,255	0%	0%	0%	0	1	6	6,254	0%	0%	0%	0	1	0	6,260	0%	0%	0%
commons-exec	55	1	0	1	1	53	0%	0%	0%	1	0	18	36	5%	100%	10%	0	1	1	53	0%	0%	0%
handlebars.java	420	1	0	1	5	414	0%	0%	0%	0	1	91	328	0%	0%	0%	0	1	0	419	0%	0%	0%
ninja	307	1	0	1	3	303	0%	0%	0%	0	1	50	256	0%	0%	0%	0	1	0	306	0%	0%	0%
Total	21,734	808	599	209	406	20,520	60%	74%	66%	583	225	4,683	16,243	11%	72%	19%	600	208	314	20,612	66%	74%	86%
AUC (Average per fold)						86%							75%							86%			

Prediction of Flaky Tests

PLRG

- Another way to statically predict flakiness is to use **lexical analysis** on the test case to extract specific **lexical patterns** on flaky tests for limited domain (network related latency, external resources not ready, file I/O, etc.)

- G. Pinto, B. Miranda, S. Dissanayake, M. d'Amorim, C. Treude, and A. Bertolino. What is the vocabulary of flaky tests? MSR 2020, pages 492–502

- Static flaky test prediction essentially becomes **text classification**

Prediction of Flaky Tests

PLRG

```java
@Test
public void testCodingEmptySrcBuffer() throws Exception {
 final WritableByteChannelMock channel = new ritableByteChannelMock(64);
 final SessionOutputBuffer outbuf = new SessionOutputBufferImpl(1024, 128);
 final BasicHttpTransportMetrics metrics = new BasicHttpTransportMetrics();
 final IdentityEncoder encoder = new IdentityEncoder(channel, outbuf, metrics);
 encoder.write(CodecTestUtils.wrap("stuff"));
 final ByteBuffer empty = ByteBuffer.allocate(100);
 empty.flip();
 encoder.write(empty);
 encoder.write(null);
 encoder.complete();
 outbuf.flush(channel);
 final String s = channel.dump(StandardCharsets.US_ASCII);
 Assert.assertTrue(encoder.isCompleted());
 Assert.assertEquals("stuff", s);
}
```

pty src buffer codec test utils standard charsets channel assert equals encoder byte buffer empty test coding empty assert allocate flush outbuf metrics dump complete wrap write flip stuff completed

Prediction of Flaky Tests

PLRG

```
@Test
public void testCodingEmptySrcBuffer() throws Exception {
 final WritableByteChannelMock channel = new ritableByteChannelMock(64);
 final SessionOutputBuffer outbuf = new SessionOutputBufferImpl(1024, 128);
 final BasicHttpTransportMetrics metrics = new BasicHttpTransportMetrics();
 final IdentityEncoder encoder = new IdentityEncoder(channel, outbuf, metrics);
 encoder.write(CodecTestUtils.wrap("stuff"));
 final ByteBuffer empty = ByteBuffer.allocate(100);
 empty.flip();
 encoder.write(empty);
 encoder.write(null);
 encoder.complete();
 outbuf.flush(channel);
 final String s = channel.dump(StandardCharsets.US_ASCII);
 Assert.assertTrue(encoder.isCompleted());
 Assert.assertEquals("stuff", s);
}
```

pty src buffer codec test utils standard charsets channel assert equals encoder byte buffer empty test coding empty assert allocate flush outbuf metrics dump complete wrap write flip stuff completed

Prediction of Flaky Tests

It achieves an F-measure of 0.95 for the prediction of flaky tests.

Table 3: Classifier performance

algorithm	precision	recall	F_1	MCC	AUC
Random Forest	**0.99**	0.91	**0.95**	**0.90**	**0.98**
Decision Tree	0.89	0.88	0.89	0.77	0.91
Naive Bayes	0.93	0.80	0.86	0.74	0.93
Support Vector	0.93	**0.92**	0.93	0.85	0.93
Nearest Neighbour	0.97	0.88	0.92	0.85	0.93

Summary

PLRG

1. Mutation Testing
    - Fundamental Hypotheses
    - Overall Process
    - Mutation Generation
    - Kill vs Alive
    - Equivalent Mutants
    - How to Kill A Mutant
    - Scalability
    - Higher Order Mutants
    - Tools

2. Test Flakiness

Next Lecture

- Mutation Testing (Homework)

Jihyeok Park
jihyeok_park@korea.ac.kr
https://plrg.korea.ac.kr